Intrusion detection in distributed systems, an approach based on taint marking

Christophe Hauser [1,2], Frédéric Tronel[1], Colin Fidge[2], Ludovic Mé[1]

[1]Supélec, CIDre Team, Rennes, France
[2] QUT, Information Security Institute, Brisbane, Australia
Email:{firstname.lastname@supelec.fr}, c.fidge@qut.edu.au

***Abstract*—This paper presents a new framework for distributed intrusion detection based on taint marking. Our system tracks information flows between applications of multiple hosts gathered in *groups* (*i.e.* sets of hosts sharing the same distributed information flow policy) by attaching taint labels to system objects such as files, sockets, Inter Process Communication (IPC) abstractions, and memory mappings. Labels are carried over the network by tainting network packets. A distributed information flow policy is defined for each *group* at the host level by labeling information and defining how users and applications can legally access, alter or transfer information towards other trusted or untrusted hosts. As opposed to existing approaches, where information is most often represented by two security levels (low/high, public/private *etc.*), our model identifies each piece of information within a distributed system, and defines their legal interaction in a fine-grained manner. Hosts store and exchange security labels in a peer to peer fashion, and there is no central monitor. Our IDS is implemented in the Linux kernel as a Linux Security Module (LSM) and runs standard software on commodity hardware with no required modification. The only trusted code is our modified operating system kernel. We finally present a scenario of intrusion in a web service running on multiple hosts, and show how our distributed IDS is able to report security violations at each host level.**

I. Introduction

As most of today's organizations rely extensively on distributed services where sets of machines are constantly and publicly exposed to the internet, firewalls, anti-virus software and Intrusion Detection Systems (IDSes) have become common additions to most of these services. However, current IDSes are mostly based on network traffic analysis and malware signatures, and little research has focussed on detecting intrusions in distributed systems. With the growing number of distributed environments and services across the internet, interest has been focussed on the extension of MAC [1], [2], [3] and information flow control [4], [5], [6] policies to distributed systems so as to control interaction between applications of multiple machines. However, such systems enforce strict policies such as Multi Level Security (MLS) and are not practical in all situations, as these can potentially break functionalities by blocking information flows. In this paper, we propose a distributed model allowing for rich policy specification and fine-grained information flow tracking. Our contributions are:

1) The design of DBlare, a distributed intrusion detection system based on taint marking.
2) An implementation of DBlare in the Linux kernel.
3) A case study of DBlare on a distributed web service running off the shelf components on commodity hardware.

II. Approach Overview

We have developed a framework to detect intrusions in distributed systems. This framework is based on information flow monitoring by attaching taint labels to containers of information (*i.e.* system objects containing information such as files, memory pages and the like). Labels contain meta-information representing pieces of information contained in tainted files. As shown in previous work [7], [8], exploiting security flaws spawns abnormal information flows between objects of the operating system, which we can detect using this framework. Most of today's companies rely extensively on networks and distributed systems running databases, web servers and other services. Such services often run proprietary software from various sources, which makes static analysis impossible unless all the software editors cooperate. Even in the case of open source software, security flaws do occur quite frequently as the code is rarely audited. Dynamic tainting of information at the operating system level, and at the network packet level allows us to detect illegal information flows amongst users and/or programs distributed in hosts, regardless of whether or not the software involved has been analysed for correctness and security. Hosts are gathered in *groups* of multiple hosts sharing a distributed information flow policy. Taint labels are propagated with every information flow between system objects of the same host *(files, sockets, shared memory)* as well as over the network through CIPSO labels. As described later in Section IV, the security policy is decentralized in each host of a *group* and verified independently by the kernel reference monitor of each host, in a distributed manner. It defines how information can flow between users, applications and hosts. Our kernel reference monitor is implemented as a Linux Security Module (LSM) and tracks access to resources at runtime, using access control hooks. Access is always permitted as we do not enforce any policy, but rather raise an alert in the case of an illegal information flow. A key difference with MAC in our approach is that we can observe indirect

information flows. For instance, when an access to a resource containing some information is granted by MAC mechanisms to a process p_a, it could then share this information with another process p_b through IPC. The information may then be written to a file f_a by p_b whereas a direct write access to the same file by p_a would have been denied by the MAC policy in the first place. Contrary to MAC mechanisms, the model we propose is able to track such indirect information flows and their legality *w.r.t.* a specified information flow policy.

III. Background: information flow control and taint analysis

In 1976, Denning introduced "a lattice model of secure information flow" [9]. She defined it as a mathematical framework suitable for formulating the requirements of secure information flow among security classes. Most of the lattice-based information flow models can be represented in Denning's framework. This formal model involves objects, processes and a set of security classes. Each object belongs to a security class, subjects are objects, and processes are the active agents responsible for all information flows. The set of security classes encompasses the concept of security classifications. Denning also introduces a "flow relation" and the "class combining operator", which together with a set of security classes forms a Lattice. In 1997, Myers and Liskov proposed a decentralized model for information flow control [10]. This model applies to systems with mutual distrust and decentralized authority. It differs from multi-level security models by allowing users to declassify information in a decentralized way and improves support for fine-grained data sharing. This model allows users to associate confidentiality and integrity labels with data and to restrict information flows based on these labels. With Decentralized Information Flow Control (DIFC), the policy is partially delegated to the individual applications [4]. It is a key difference with Mandatory Access Control (MAC) systems, where an administrator sets a system-wide policy. Current decentralized information flow control (DIFC) models can be categorized into three types: language level, operating system level and architecture level [11]. Language level solutions [10], [12] rely extensively on type system changes and modify the program structures. Such solutions provide no guarantees against security violations on system resources (such as files and sockets). Operating system level solutions [4], [5], [6] rely on page mappings and are inefficient to accurately monitor information flows into applications as those do not have access to inner data structures [11]. Architecture level solutions [13], [14] are able to track data labels within applications but require trusted software to manage the labels. Laminar [11] is a hybrid solution combining language level and operating system level information flow control, but still requires modifications in the code of the programs. Flume, Asbestos and Histar [4], [5], [6] are implementations of distributed information flow control at the operating system level. Flume [4] has been implemented in Linux and uses the standard operating system abstractions commonly found on UNIX systems (processes, pipes, etc.). In Flume, processes are confined according to a flow control policy. Histar [5] is an operating system aiming to minimize the amount of code that must be trusted. It provides a secure operating system using mostly untrusted user-level libraries (the only fully trusted code being the kernel). It uses Asbestos [6] labels on six OS level object types (threads, address spaces, segments, gates, containers and devices). Recent works regarding the monitoring of information flows include Panorama [15], TaintCheck [16] and TaintDroid [17]. Panorama is a system-wide information flow tracking model based on dynamic taint analysis. TaintCheck dynamically taints incoming data from untrusted sources (*e.g.* network) and detects when tainted data is used in any way that could be an attack. Both use full system emulation at the instruction level to provide very fine-grained approaches. However, the main limitation of such instruction-level models is a very high penalty in terms of performances; a slowdown of 20 times in average when using Panorama, and a slowdown of 1.5 to 40 times when using TaintCheck are to be expected, according to their respective authors. TaintDroid [17] is an information flow tracking system for realtime privacy monitoring on smartphones. It is based on taint marking at four different levels of granularity, respectively at the variable, message, method and file levels. TaintDroid has a performance overhead of 14% on the CPU. However, TaintDroid is focussed on the Android platform using the Dalvick interpreter and therefore it does not apply to native applications, which represent most of the software present on standard desktop and server operating systems. Our approach uses dynamic tainting of OS level objects (processes, files, sockets, IPC, memory mappings and pipes) to track both native and interpreted[1] applications and does not require any modification or instrumentation of the code nor architecture emulation, resulting in a reasonable performance overhead (for brevity, this aspect is not covered in this article). As opposed to existing approaches, where information is most often represented by two security levels (low/high, public/private *etc.*), our model identifies each piece of information within a distributed system, and defines their legal interaction in a fine-grained manner.

IV. Distributed intrusion detection

In previous work [7], [8] we have been using taint marking techniques along with an information flow policy to detect intrusions and confidentiality violations at the operating system level. This current work extends the previous model to allow for intrusion detection in distributed systems.

A. Overview of Blare

Blare attaches labels to containers of information, *i.e.* system objects involved in information flows such as files, processes, memory mappings, socket buffers and others. We also refer to those labels as *tags*. Information sources, such as data contained in files, are tainted by initially attaching *information tags* (also called *itags*) to their containers. *Information tags* contain meta-information describing the origin

[1] We track the actions of the virtual machine in the case of interpreted code.

of information as well as its nature, active code or passive data, respectively represented by two sets $\mathcal{X}$ and $\mathcal{I}$. Code is considered active when it is being executed by a process, it is otherwise considered as passive (stored) data. This distinction follows Denning's assumption that "processes are the active agents responsible for all information flows". *Information tags* are sets of meta-information describing the content of containers, namely any combination of elements of $\mathcal{I}$ and $\mathcal{X}$, that we note $\wp(\mathcal{I} \cup \mathcal{X})$[2]. The execution of code is represented formally by the following function, mapping passive data to running code:

$$exec : \mathcal{I} \rightarrow \mathcal{X}$$

When a piece of information $i \in \mathcal{I}$ is executed by a process (*e.g.* the content of an executable file) its *information tag* is updated with $x = exec(i), x \in \mathcal{X}$. Elements of $\mathcal{X}$ in a process's information tag indicate which code[3] is currently being run, whereas elements of $\mathcal{I}$ indicate passive data[3] which have been read by this process. Elements of X in other containers keep tracks of which processes (running tainted code) wrote information in the container. Information tags are updated after every information flow, and are a maximal estimation[4] of the current content of their associated container at any given point in time. Figure 1 summarizes the tainting rules used in our model. *Policy tags* are sets of sets, (*i.e.*

Operation	$i \in \mathcal{I}$	$x \in X$
`Read`	taint	check legality & discard
`Write`	taint	taint
`Execute`	taint with $x = exec(i)$	discard

Fig. 1. Tainting rules: when a process reads, executes or writes from/towards a container, distinct tainting rules apply. *Taint* means that the destination process or container gets tainted by the meta-information. *Discard* means the destination process or container does not get tainted by the meta-information.

elements of $\wp(\wp((\mathcal{I} \cup \mathcal{X})))$ each describing which combinations of information can legally be contained in the containers they are attached to. The host's information flow policy is thereby distributed in all the containers of the system. An information flow towards a container is legal if and only if its updated *information tag* is included in (at least) one of the sets of its *policy tag* after the flow occurred (except on `read` operations involving elements of $\mathcal{X}$, in this case the legality check is performed on the fly, and only elements of $\mathcal{I}$ are kept in the updated tag, as shown in Figure 1.). *Policy tags* of processes are dynamically determined at execution time as the intersection of the policy attached to the program's file and the policy attached to the current user. This point will be further detailed in the following section.

[2]Powerset $\wp(A)$ denotes all the subsets of A.

[3]It is an overestimate based on the potential information hold by the process.

[4]After each information flow occurs, the updated *information tag* is the union of the source's *information tag* and the destination's *information tag*. This conservative approach may be an overestimation of the actual content, but it is necessary as it is unpractical to observe actual information flows [14].

B. Distributed model

In our current research, we have now extended the previous model in order to detect intrusions in distributed systems composed of multiple hosts gathered in *groups*. Hosts of the same *group* share a common information flow policy. The policy is distributed in each host at the container level. *Tags* are carried over the network using CIPSO[5], an IETF draft proposing a "Commercial IP Security Option". CIPSO is a security option of IPv4 packets allowing to carry security labels between hosts.

Trusted hosts run our modified kernel, and communicate over secure channels (*e.g.* VPN, IPSec etc.) to ensure the integrity of labels is kept. Each host of a *group* is identified by a unique id $h_k \in \mathcal{H}$. For any given host h_k, we define $\mathcal{I}_k$ to be the set of all passive data managed by this host, and $\mathcal{X}_k$ the image of $\mathcal{I}_k$ through the $exec$ function, *i.e.* the code originating from this host which may be executed by processes on any host. $\mathcal{I}_k$ and $\mathcal{X}_k$ are partitions of respectively $\mathcal{I}$ and $\mathcal{X}$ representing all the information of the *group*:

$$\mathcal{I} = \bigcup_{h_k \in \mathcal{H}} \mathcal{I}_k \wedge \mathcal{X} = \bigcup_{h_k \in \mathcal{H}} \mathcal{X}_k$$

Recall that *information tags* are sets of elements in $\wp(\mathcal{I} \cup \mathcal{X})$, identifying passive data and active code residing in containers. The originating host of an element of $(\mathcal{I} \cup \mathcal{X})$, *i.e.* the host managing a given piece information, is determined by the following relation:

$$Host : (\mathcal{I} \cup \mathcal{X}) \rightarrow \mathcal{H}$$

An information flow towards a container is legal if and only if its *information tag* T_i is included in at least one of the sets of its *policy tag* T_p, which we note $T_i \sqsubseteq T_p$. This applies to all kinds of containers (*i.e.* processes as well as passive containers). We define the boolean relation $Legal$ as follows:

$$Legal(T_i) \Leftrightarrow T_i \sqsubseteq T_p$$

$$T_i \sqsubseteq T_p \Leftrightarrow \exists A \in T_p | T_i \subseteq A$$

Processes are responsible for all information flows. For each set of the *policy tag* of a given process:

- Any element $a \in \mathcal{X}$, allows communication with other processes executing a on any other host of the group.
- Any element $a \in \mathcal{I}$ allows information a to flow from another host of the group towards the current process.

The information flow policy $\mathbb{P}_{group}$ of a group of hosts $(h_1, \cdots, h_N)$, identified at each host's level is defined as

$$\mathbb{P}_{group} = (\mathbb{P}_{h_1}, \cdots, \mathbb{P}_{h_N})$$

The local information flow policy on any host h_i, is expressed independently for users, programs and passive containers (*e.g.* files), and is specified in the *policy tags* of containers[6]. It is defined by the triplet $\mathbb{P}_{h_i} = (\mathbb{P}_{\mathcal{C}_{h_i}}, \mathbb{P}_{\mathcal{U}_{h_i}}, \mathbb{P}_{\Pi_{h_i}})$ where:

[5]`https://tools.ietf.org/html/draft-ietf-cipso-ipsecurity-01`

[6]Policy conflicts are resolved manually (automatic resolution is left for future work).

- $\mathbb{P}_{\mathcal{C}_{h_i}}$ is the set of all the policy tags attached to passive containers (mostly files) on host h_i.
- $\mathbb{P}_{\mathcal{U}_{h_i}}$ is the policy attached to local users on host h_i.
- $\mathbb{P}_{\Pi_{h_i}}$ is the policy attached to executable code on host h_i.

Processes run code on behalf of a user. Their policy tags are determined dynamically by $\mathbb{P}_{U_{h_i}}$ and $\mathbb{P}_{\Pi_{h_i}}$, at execution time, as the intersection of the policy attached to the user on behalf of which the program is being executed, and the policy attached to the executed program.

C. Distributed security tokens

Information tags have dynamic sizes, and can therefore require variable amounts of space depending on the situation. Due to size restrictions on IPv4 options, it may not always be possible to directly map *information tags* to CIPSO options on network packets. Therefore, we have introduced a distributed security token protocol allowing hosts of a *group* to exchange security labels in a peer to peer fashion, as shown in Figure 2. Security tokens are images of *information tags* through the relation H as follows, where Θ is the set of all possible CIPSO tokens:

$$H : \wp(\mathcal{I} \cup \mathcal{X}) \rightarrow \Theta$$

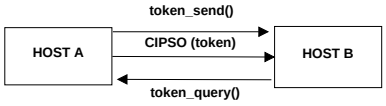

Fig. 2. P2P token exchange: host A labels network packets with a CIPSO option containing a security token. If Host B does not find it in its local cache, it asks host A for resolution using the protocol defined in this section. Note that H is **not injective** since the size of θ is smaller than the potential size of information tags.

We use tokens as security labels on network packets so as to carry taint information between the multiple hosts of a *group*. Recall that *information tags* are dynamic: their content is updated after every information flow. Therefore, processes often need to update the labels they attach to network packets: after every information flow, if the *information tag* of the process has changed, a new token is created for this process, corresponding to its new *information tag*. Every host maintains a local cache of sent and received tokens for each remote host of the *group*, as shown in Figure 3. We note $sent_{h_A}[h_B]$ and $recv_{h_A}[h_B]$, respectively the cache of tokens sent to host h_B and the cache of tokens received from host h_B on host h_A. The sizes of sent and received caches are equal and noted ℓ. Caches contain $\langle key : value \rangle$ pairs in $(\Theta \times \wp(\mathcal{I} \cup \mathcal{X}))$. Token resolution and token creation (described below) ensure that for each pair of hosts (h_A, h_B), $sent_{h_B}[h_A]$ is synchronized with $recv_{h_A}[h_B]$. Token resolution is performed over an alternate secure channel, using unlabeled packets (*i.e.* no CIPSO options). Possible operations on both caches are: creating a new entry (C), overwriting an existing entry (O) and reading an existing entry (R).

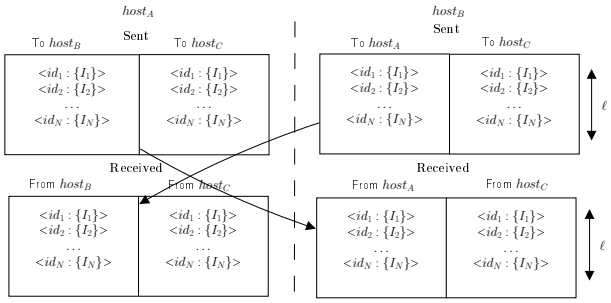

Fig. 3. Distributed token protocol. Hosts maintain caches of sent and received tokens for each host of their group.

a. Token resolution: when a process receives a network packet labeled with a security token, it needs to *resolve* it in order to be able to append the appropriate taint data to its *information tag*. Token resolution is defined by the following relation:

$$resolve : \Theta \rightarrow \wp(\mathcal{I} \cup \mathcal{X})$$

Token resolution can be done directly if the token is in the local reveived tokens cache. Overwise, it is necessary to query the remote host using the protocol defined below.

b. Token creation: when a process on host h_{local} updates its *information tag*, the following actions are necessary before sending data to any destination host h_{dest}.

1) Create a new token $tk_{new} = H(itag)$ where $itag$ is the current *information tag* of the process, and H is a cryptographic hash function.
2) Read local cache entries (R) and check for collisions[7] with tk_{new}: $sent_{h_{local}}[h_{dest}][tk_{new}]$ may already exist as a key for a different *information tag*.
3) In case of collision, overwrite (O) existing values and set $flag$ to `O_REPLACE`.
4) Otherwise, create (C) tk_{new} in $sent_{h_{local}}[h_{dest}]$ and set flag to `O_NEW`.
5) Send token to remote host (using the protocol defined below and the appropriate flag).

c. Token exchange protocol: hosts of a *group* exchange tokens using a protocol based on the two following operations:

- Function $token\_query(token, host)$: query $host$ about $token$. The remote host replies with $token\_send$ and sets a flag to either `O_NEW` or `O_REPLACE`. When `O_REPLACE` is set, a previous cache entry with the same key already exists and must be replaced. Otherwise, it is a new entry.
- Function $token\_send(token, flag, host)$: send the pair $(token, H(token))$ to $host$ with $flag$ in {`O_NEW`, `O_REPLACE`}.

V. Implementation

The reference monitor for this model has been implemented in the Linux kernel version 3.2 as a LSM module (Linux

[7] H is a hash function, however collisions may occur in some rare cases.

Security Module), and is available as an open source project[8]. Labels are stored in the extended attributes of the filesystem, in the `security` namespace, as arrays of integers of variable size (*i.e.* these arrays have dynamic sizes). *Information tags* in kernel memory have been implemented as doubly linked lists of integers. *Policy tags* have been implemented as linked lists of binary trees (red black trees), one binary tree per set, *policy tags* being sets of sets. Checking the legality of an information flow is an operation whose complexity is in $O(k \times \ell \times log_2(n))$, where k is the length of the involved *information tag*, l is the number of subsets of the policy and n is the maximum size of the sets of the policy. Information flows are tracked between files, memory mappings, pipes, message queues, shared memory and sockets (both `AF_UNIX` and `AF_INET`). Network packets are labeled using the Netlabel subsystem, an API in the Linux Kernel implementing CIPSO (Commercial IP Security Option). CIPSO option size is limited to 40 bytes (320 bits), which limits the amount of meta-information we are able to carry over the network. *Information tags* are sets of 32 bit integers, therefore CIPSO options can only contain less than 10 *information tag* elements per network packet. In order to overcome this limitation, we have designed a distributed security token management protocol, allowing any host of a *group* to securely exchange security labels. However, for the sake of simplicity, our current implementation labels network packets with *information tags* of up to 320 distinct values, by using fixed-size bitmaps. Our experiments in the following section have been developed as as a proof of concept on this current implementation, as those only require a limited number of distinct labels. Further experiments will be conducted in future work, with a full implementation, in order to measure the impact of our distributed token system on the performances of the operating system (*i.e.*, CPU, memory and network latency).

VI. Experiments

The following section describes our experiments. We have set up an attack scenario targeting a *group* of trusted hosts running our modified kernel. This *group* is composed of three hosts: a web server, a database server and a client, all three connected to the same Virtual Private Network (VPN). The web server (Apache) hosts two websites, isolated in two virtual hosts www_1 and www_2 (Apache vhosts[9].) The database server (PostgreSQL) hosts two databases, storing data of the two virtual hosts: db_1 stores information related to www_1, and db_2 stores information related to www_2. Connections to www_1 are allowed from the outside. Connections to the other hosts of the VPN and to www_2 are forbidden from the outside. The following shows how it is possible with our intrusion detection model to detect illegal information flows between hosts caused by an intrusion. We used Debian Squeeze virtual guests running as KVM [18] instances. The two websites run Wordpress. The website www_1 runs the e-commerce plugin Foxypress[10]. We used the version 0.4.2.2 of this plugin, which is vulnerable to an upload exploit (EDB-ID: 19100)[11]. This vulnerability allows for arbitrary file upload and remote code execution.

A. Scenario

As shown on Figure 4, we labeled all the files of www_1 and www_2 as well as the PHP5 dynamic library (used by apache to interpret PHP code) with distinct *information tags* on the web server. On the database server, we labeled the PostgreSQL binary as well as two tables on each database. We could label information at the table level by using the option `default_with_oids = on` in PostgreSQL's configuration file. Object identifiers (OIDs) are used in PostgreSQL as primary keys for system tables, as well as user-created tables when using this option. Each table in PostgreSQL is mapped to a file named after its OID. Thus, we could label the files related to the supervised tables.

Host	Files	*Itag*	Execution
Web server	www_1	i_1	x_1
	www_2	i_2	x_2
	`libphp5.so`	i_{php}	x_{php}
	`apache2`	i_a	x_a
Database server	db_1: `wp_users`	i_{u_1}	x_{u_1}
	db_1: `wp_posts`	i_{p_1}	x_{p_1}
	db_2: `wp_users`	i_{u_2}	x_{u_2}
	db_2: `wp_posts`	i_{p_2}	x_{p_2}
	`postgres`	i_{pg}	x_{pg}

Fig. 4. Labels on files

By default, both Apache and PostgreSQL create a new process for each connection. Recall the exec function from the previous section. When a process executes a binary file (or the content of a dynamic library) labeled with i_k, its *information tag* is set to $x_k = exec(i_k)$. Therefore, both Apache and PostgreSQL processes always have their *information tags* initialized to respectively $x_a = exec(i_a)$ and $x_{pg} = exec(i_{pg})$. We used the following *policy tag* for both Apache and PostgreSQL processes: $P = \{\{x_a, x_{pg}, x_{php}, i_1, i_{u_1}, i_{p_1}\}\{x_a, x_{pg}, x_{php}, i_2, i_{u_2}, i_{p_2}\}\}$. Such a policy makes it illegal for any process running Apache or PostgreSQL to hold information from both websites simultaneously, or to run any code other than Apache and PosgreSQL binaries and libphp5. When an external visitor visits www_1, the web server creates a new process for this connection and reads files labeled with i_1. It also maps `libphp5.so` in executable memory pages which taints the process with x_{php}. It queries the database server. The database server forks a new process and reads information from db_1. At this stage, the *information tag* of the PostgreSQL process is tainted with $S_1 = \{x_a, x_{pg}, x_{php}, i_1, i_{u_1}, i_{p_1}\}$. After the PostgreSQL process has responded to the Apache process, both processes have equal *information tags*, as each process

[8] http://blare-ids.org

[9] From `http://www.apache.org`: the term Virtual Host refers to the practice of running more than one web site on a single machine.

[10] `www.foxy-press.com`

[11] `http://www.exploit-db.com/exploits/19100/`

labels network packets with a CIPSO option containing its *information tag* (in a bitmap, as described in Section V). When an internal host connects to the internal virtual host www_2, similar interactions happen between the hosts, and the *information tags* of both processes handling the connection are tainted with $S_2 = \{x_a, x_{pg}, x_{php}, i_2, i_{u_2}, i_{p_2}\}$. In both cases, information flows are legal, and so no alert is raised, because *information tags* are subsets of the *policy tags* in both containers: $S_1 \sqsubseteq P \wedge S_2 \sqsubseteq P$.

B. Attack

The following attack leaks information from the private web site www_2 located on the intranet. The attacker runs the upload exploit on the Foxypress plugin on www_1 and injects a malicious PHP file on the web server. We used Metasploit[12] to run the attack. After injecting the file, the running web server process's *information tag* was equal to $S_1 = \{x_a, x_{pg}, x_{php}, i_1, i_{u_1}, i_{p_1}\}$, and so was the *information tag* of the malicious PHP file. From there, any illegal action triggered an alert:

- Executing the malicious PHP file, which taints[13] the process's *information tag* with $exec(S_1) = \{x_1, x_{u_1}, x_{p_1}\}$ is illegal, as $exec(S_1) \not\sqsubseteq P$
- Querying the database server to access data from www_2, which taints the process's *information tag* with $S_3 = \{x_a, x_{pg}, x_{php}, i_1, i_{u_1}, i_{p_1}, i_{u_2}, i_{p_2}\}$ is illegal as well, as $S_3 \not\sqsubseteq P$.

Information tags are carried over the network through CIPSO labels, therefore both the web server and the database server raise an alert in the case of illegal information flow, as both servers are affected by the attack: data from the database server leaks, and the web server runs arbitrary code.

VII. Conclusion

Distributed systems are used all over the internet to provide services involving a number of different software connected together. New security flaws are discovered regularly, which makes it challenging for system administrators to keep such systems secured at all times. In this paper, we presented a novel approach of intrusion detection in distributed systems based on taint marking along with an information flow policy. The policy for a group of hosts is distributed in each host at the container level (*i.e.* files, processes *etc.*). CIPSO labels are attached to network packets in order to carry security information between hosts. Our kernel reference monitor has been implemented as a Linux security module and uses efficient data structures for runtime tainting (for the sake of brevity, we have not include a performance analysis in this article). It is used on each host of a group, and alerts are reported at each host's level with respect to its local information flow policy. We have been running experiments involving a web service distributed on several hosts, and we presented an example of application showing how our system is able to detect an attack targeting private information by exploiting a vulnerability on a web server.

[12] `http://www.metasploit.com`

[13] Apache maps PHP files in executable memory pages (`PROT_EXEC`), like it does with dynamic libraries.

References

[1] J. M. McCune, T. Jaeger, S. Berger, R. Cáceres, and R. Sailer, "Shamon: A system for distributed mandatory access control," in *Proceedings of ACSAC*, 2006, pp. 23–32.

[2] A. Hernandez and F. Nielson, "Enforcing mandatory access control in distributed systems using aspect orientation," in *21st Nordic Workshop on Programming Theory – NWPT2009*, 2009, pp. 62–64.

[3] R. Wu, G.-J. Ahn, H. Hu, and M. Singhal, "Information flow control in cloud computing," in *CollaborateCom'10*, 2010, pp. 1–7.

[4] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *Proceedings of the 21st Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[5] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 263–278.

[6] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2005, pp. 17–30.

[7] Stéphane Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong, "Information flow control for intrusion detection derived from mac policy," *Proceedings of the IEEE International Conference on Computer Communications (ICC)*, 2011.

[8] C. Hauser, F. Tronel, J. F. Reid, and C. J. Fidge, "A taint marking approach to confidentiality violation detection," in *10th Australasian Information Security Conference (AISC 2012)*, J. Pieprzyk and C. Thomborson, Eds. RMIT University, Melbourne, VIC: Australian Computer Society, January 2012. [Online]. Available: http://eprints.qut.edu.au/47263/

[9] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[10] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 129–142, 1997.

[11] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: practical fine-grained decentralized information flow control," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[12] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410–442, 2000.

[13] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 243–254.

[14] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware enforcement of application security policies using tagged memory." in *Proceedings of OSDI*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 225–240.

[15] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. ACM, 2007, pp. 116–127.

[16] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[17] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. USENIX Association, 2010, pp. 1–6.

[18] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, no. 166, Feb. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1344209.1344217